Chapter 1

Object Oriented Programming

1.1 Object oriented approach

Definition 1.1.1: Object Oriented Programming

Object oriented programming (OOP) is a imperative programming programming paradigm where instructions are grouped together with the part of the state they operate on.

The main **structural components** of all systems are

- Objects: Object is something that takes up space in the real or conceptual world with which somebody may do things. An object is an instance of a class. An object has three main characteristics:
  - Name (or ID): Unique identifier for the object.
  - State: Attributes that describe the object.
  - Operations (or behavior): Actions that the object can perform.
- Class objects: Class is the blueprint of an object.

Main **characteristics** of the approach are

- Encapsulation: The mechanism of hiding the implementation of the object. Only the necessary details are exposed to the user.
- Abstraction: A principle which consists of ignoring the aspects of a subject that is not relevant for the present purpose. It is the process of simplifying complex systems by modeling classes based on the essential properties and behaviors an object must have.
- Inheritance: Mechanism by which one class can inherit attributes and methods from another class. Making new classes from existing ones.
- Polymorphism: Ability to present the same interface for different data types. Same function name but different signatures being used for different types.

1.2 Class

Definition 1.2.1: Class Diagram

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

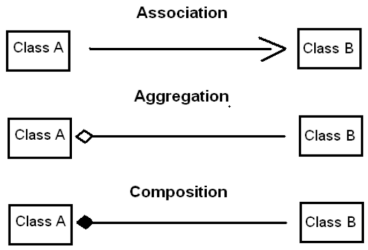
Elements of a **class**:

- Class Name (ID Class): The name of the class. Nouns associated with the textual description of a problem. Singular.

- Attributes: The properties or characteristics of the class. Types include Real, Integer, Text, Boolean, Enumerated, etc.
- Operations (Methods): The functions or operations that can be performed on the class. Behaviors of the class.

Relations between classes:

- Association: A relationship between two classes that indicates how objects of one class are connected to objects of another class.
- Aggregation: A special type of association that represents a "whole-part" relationship between classes.
- Composition: A stronger form of aggregation that implies ownership and a whole-part relationship where the part cannot exist independently of the whole.
- Generalization: A relationship between a more general class (superclass) and a more specific class (subclass) that indicates inheritance.



1.2.1 Class and Method in Python

```python
class Person:
    pass # An empty class definition

p = Person()
print(p)  # Output: <__main__.Person object at 0x...>
```

Listing 1.1: Empty class definition in Python

```python
"""Define class with method"""
class Person:
    def speak(self):
        print("Hello!")

"""Create object and call method"""
p = Person()
p.speak()  # Output: Hello!
```

Listing 1.2: Method

```python
"""
The __init__ method is a special method in Python classes.
It is a method that Python calls when you create a new instance of this class.
"""
class Person:
    def __init__(self, name):
        self.name = name  # Initialize the name attribute
    def speak(self):
        print('Hello, my name is', self.name)

p = Person('Carlos')
p.speak()  # Output: Hello, my name is Carlos
```

Listing 1.3: Attributes and __init__ method

Note:

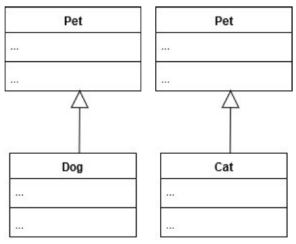
The first argument of every class method, including init, is always a reference to the current instance of the class. By convention, this argument is always named self.

```
class Pet(object):
    def __init__(self, name, species):
        self.name = name
        self.species = species
    def getName(self):
        return self.name
    def getSpecies(self):
        return self.species
    def __str__(self):
        return "%s is a %s" % (self.name, self.species)
```

Listing 1.4: Class Pet with attributes and methods

```
class Dog(Pet):

"""Dog inherits from Pet"""

    def __init__(self, name, chases_cats):
        # Call the parent class (Pet) constructor
        Pet.__init__(self, name, "Dog")
        self.chases_cats = chases_cats

    def chasesCats(self):
        return self.chases_cats

class Cat(Pet):

"""Cat inherits from Pet"""

    def __init__(self, name, hates_dogs):
        # Call the parent class (Pet) constructor
        Pet.__init__(self, name, "Cat")
        self.hates_dogs = hates_dogs

    def hatesDogs(self):
        return self.hates_dogs
```

Listing 1.5: Inheritance



Example 1.2.1 (Inheritance)

myPet = Pet("Boby", "Dog")
myDog = Dog("Rex", True)

- isinstance(myDog, Pet) # Returns: True
- isinstance(myDog, Dog) # Returns: True

- isinstance(myPet, Pet) # Returns: True
- isinstance(myPet, Dog) # Returns: False

1.2.2 Access Modifiers

There are three types of access modifiers in Python:

- Public: Members (attributes and methods) declared as public are accessible from anywhere.
- Protected: Members declared as protected are accessible within the class and its subclasses. In Python, this is indicated by a single underscore prefix (e.g., _attribute).
- Private: Members declared as private are accessible only within the class itself. In Python, this is indicated by a double underscore prefix (e.g., __attribute).

```
class Person:
    def __init__(self, name, age):
        self.__name = name      # Private attribute
        self.__age = age        # Private attribute

p = Person("David", 23)
p.__name  # This will raise an AttributeError
```

Listing 1.6: Private Attributes

```
class Person:
    def __init__(self, name, age):
        self.name = name      # Public attribute
        self.age = age        # Public attribute

p = Person("David", 23)
p.name  # This will work fine and return "David"
```

Listing 1.7: Public Attributes

```
class Person:
    def __init__(self, name, age):
        self._name = name      # Protected attribute
        self._age = age        # Protected attribute

p = Person("David", 23)
p._name  # This will work, but it's discouraged to access protected members directly
```

Listing 1.8: Protected Attributes

```
class Person:
    def __init__(self, money=0, energy=100):
        self.money = money
        self.energy = energy

    def work(self, hours):
        if self.energy >= hours * 10:
            self.money += hours * 10 # Assume earning $10 per hour of work
            self.energy -= hours * 10
            print (f"Worked for {hours} hours. Money increased to ${self.money}, energy
                decreased to {self.energy}.")
        else:
            print("Not enough energy to work.")
```

Listing 1.9: Example Usage